POWER PROGRAMMING

Memory Management and Mapped Files in Win32

BY RAY DUNCAN

The segmented architecture of Intel 80*x*86 microprocessors has been the target of much scorn by aficionados of RISC or 680*x*0 CPUs, but this derision is largely undeserved. The Intel protected-mode memory architecture, wherein a virtual address is composed of a handle (selector) and an offset, is an excellent substrate for a robust, multitasking environment, because it allows processes to be built out of memory "objects" that can be shared or segregated as distinct units without regard to their location in physical memory. Moreover, the critics conveniently ignore the segmentation in their own backyards: Some RISC processors (such as the original IBM "America" RISC CPU) are burdened with segmentation under another name, while Macintosh applications are constructed from code resources that are not allowed to exceed 32K.

The real problem with segmentation on Intel processors was not so much that it existed, but that segments were limited to 64K. Because of this, the address arithmetic in large 80*x*86 programs with code or data spanning multiple segments was very complicated, and the performance of such large programs was correspondingly poor.

When Intel designed its 80386 processor, segments were redefined in a way that allowed them to grow to 4GB, but this introduced a new problem: Segment-based virtual memory management was no longer feasible. Imagine a program's execution being suspended for several seconds or more while a segment of several megabytes was read in from the swap file, just to let the program access a single variable in that segment! Consequently, Intel added another layer of virtual-memory management: the hardware paging unit.

When an Intel 80386 or 80486 is running in protected mode with paging enabled, there are two layers of virtual address translation. The selector and offset visible at the programming level are converted by the hardware segmentation unit—with the aid of descriptor tables that are maintained by the operating system—into a 32-bit linear address. The linear address, in turn, is remapped by the hardware paging unit with the aid of page tables and a page directory (again maintained by the operating system) onto a 32-bit physical address identifying a 4K page of RAM and an offset within that page, as shown in Figure 1. The result is a 64-terabyte virtual address space superimposed on a 4GB physical address space.

A look at the evolution of memory management in Windows, especially the new mapped-file functions designed for the 32-bit flat, paged memory model.

As you can see, interposing a paging layer eliminates nearly all of the problems with segmented memory. Every application can be a "small" model, with a single, pure, shareable code segment and an instance data segment ranging in size from 1 byte to 4GB—while the operating system is free to allocate and swap physical memory in nice, uniform, 4K chunks.

One of the interesting aspects of the two-tiered virtual memory management architecture of the 80386 and 80486 is that it can readily mimic the flat 32-bit memory model of CPUs like the Motorola 68040 and the VAX. You can't ever disable segmentation on an Intel processor—segment registers must always be loaded with valid selectors—but you can choose to ignore it. This is accomplished very simply by creating two descriptors, one for code and one for data, each mapping to the same giant memory segment at linear addresses 0 to 4GB. Application code is mapped to addresses at the bottom of the segment, and operating system code to addresses at the top of the segment (see Figure 2). The CPU's segment registers are loaded with the code or data selector as appropriate when the system starts up and is never touched again. All memory references, jumps, and calls are near; memory allocation, swapping, segregation of code from data, protection of operating system code, and the separation of one process's address space from another are all handled by manipulation of the page tables and page directory. Both OS/2 2.0 and NT/Win32 use this flat-memory model when running 32-bit applications.

Why did the designers of OS/2 2.0 and NT/Win32 decide to forgo the significant benefits of the segmented paged-memory model and adopt the flat model with its much smaller virtual address space? The reasons are partly economic and partly technical. From an economic standpoint, there are many expensive, exotic, hardware-hungry applications currently available only on workstations or superminicomputers. IBM and Microsoft hope to make porting of those applications to

Addressing Memory Using Page Tables

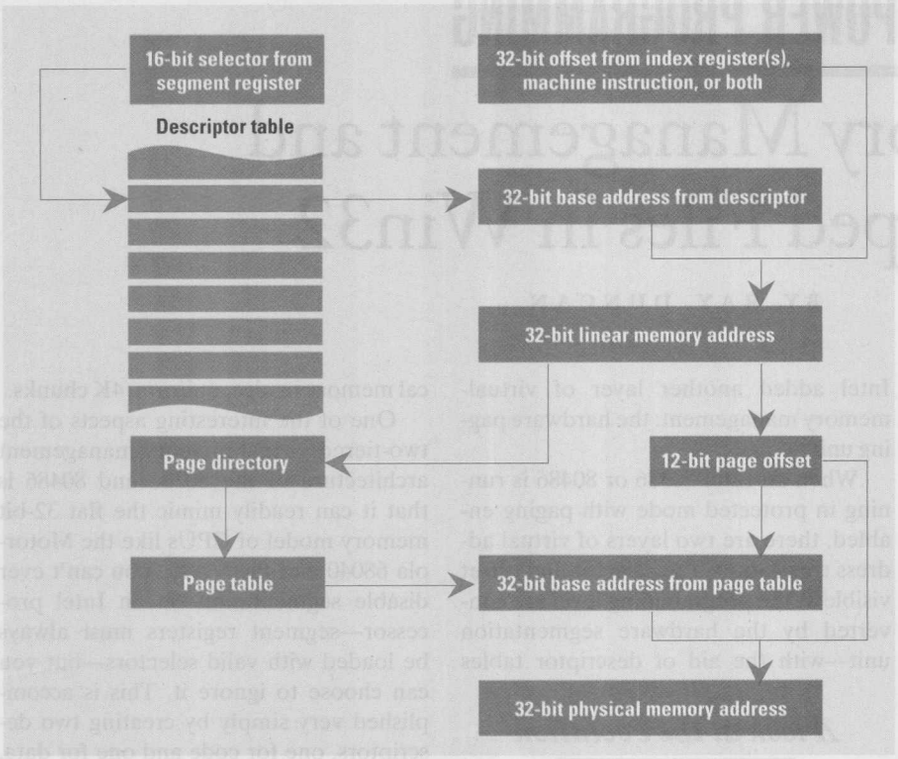


Figure 1: **Memory addressing in 32-bit protected mode on the Intel 80386 and 80486 processors. The virtual address seen at the machine-code level, composed of a selector and offset, is translated by the hardware segmentation unit into a 32-bit linear address. The linear address is converted by the hardware paging unit into a 32-bit physical address.**

OS/2 2.0 and NT/Win32 easier—and consequently much more attractive—by hiding segmentation (incidentally opening up a new and far less price-sensitive market for systems software that until now has been the exclusive province of companies like Sun, Hewlett-Packard, and Silicon Graphics).

The technical reason for this decision is that Microsoft and IBM are hedging their bets on the future of the microprocessor marketplace. Although the Intel 80*x*86 processors still dominate the installed base and their performance has improved dramatically with each successive generation, there is no guarantee that the Intel 80*x*86 architecture won't simply run out of steam. By abandoning segmentation, Microsoft and IBM have simplified the task of porting OS/2 or NT and their applications to other, non-Intel processors.

THE WIN32 MEMORY MANAGEMENT API

For an experienced Windows programmer who has thought about the underlying memory models, converting a 16-bit Windows application for Win32/NT is a rather spooky business. The fundamental Windows memory management API—GlobalAlloc(), GlobalLock(), GlobalReAlloc(), GlobalUnlock(), GlobalFree()—was originally designed for the paragraph-oriented, based-addressing architecture of the Intel 8086 real mode. It survived the transition to the Intel 80286's 16-bit protected mode quite gracefully with the substitution of selectors for handles or paragraph addresses. But source code that was written for the 16-bit Windows memory API (Win16) compiles and runs fine on a MIPS R4000 RISC CPU running Win32/NT even though the MIPS processor has no selectors at all! How can this be? This is information hiding with a vengeance!

Most of the portability between the Win16 and Win32 memory models and APIs is accomplished at the source code level with TYPEDEFs and #DEFINEs in the Windows header files. Far memory references, far pointers, and far jumps or calls are transformed cheerfully and silently to 32-bit near equivalents by the simple expedient of redefining FAR as a no-op. All data types (such as LPSTR, that are built on top of the FAR keyword with TYPEDEFs thus collapse to near data types by default. Similarly, function prototypes that were previously declared with the FAR keyword or with FAR-based TYPEDEFs as being entered through far calls and exited through far returns devolve automatically to near functions. The arguments and results of the Windows memory management functions are affected by these redefinitions like everything else.

For example, in Windows 3.*x*, the function GlobalLock() accepts a 16-bit handle and returns a selector and offset, while in Win32 the same function accepts a 32-bit offset and returns the same 32-bit offset. (In other words, in Win32 the function does absolutely nothing.) In both environments, GlobalLock() is prototyped as:

```
LPVOID APIENTRY GlobalLock(HANDLE)
```

The differences in the underlying memory models are hidden from the application within the definitions of the LPVOID, HANDLE, and APIENTRY data types (as shown in Figure 3).

Another implication of this disappearance of far pointers in Win32 is that the Localxxx Windows API functions become mere aliases for their Globalxxx counterparts. The fact that this informa-

The Flat-Memory Model

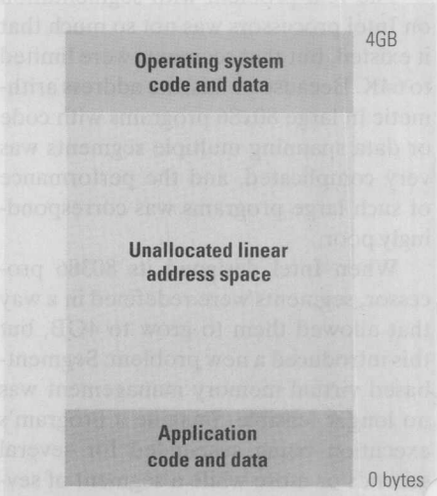
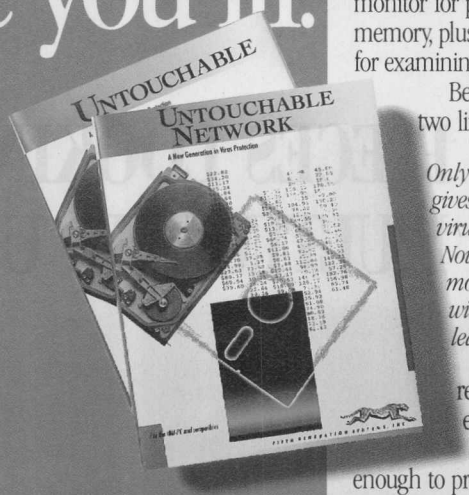


Figure 2: **The flat-memory model of OS/2 2.0 and NT/Win32 on Intel 80386 and 80486 processors.**

Paying again and again for virus protection is enough to make you ill.



Frankly, paying anything at all is pretty nauseating.

But that's nothing compared to the feeling you get in your stomach when a drive full of data goes down the drain.

So you pay.

Then you pay again. For upgrades. Or quarterly "updates."

But even that doesn't buy you much peace of mind. Because each new day brings an average of six new viruses into the world.

Which means all anti-virus programs are inherently obsolete.

Except one.

Introducing Untouchable.™ The only software in the world that gives you 100% protection.

Today.

And tomorrow.

Like other anti-virus software, Untouchable is equipped with a TSR monitor for patroling your system memory, plus a scanner/remover for examining the files on your disk.

Between them, these first two lines of defense can recognize and obliterate over 1000 of the little buggers—enough to protect you from 95% of the potential carnage.

Only Untouchable Network gives you centralized virus protection. Now you can install, monitor and control without having to leave your seat.

If you find that statistic reassuring, then you probably like the odds in Russian Roulette.

If you don't, you'll want to know that Untouchable is equipped with a unique *third* line of defense, which has been mathematically proven to be, well, untouchable.

Instead of looking for viruses, this third line of defense looks for *changes in your executable and system files.*

If the change is but a single byte, Untouchable will spot it and, using our patent-pending recovery technique, reconstruct the file to its original state.

The process is known as Generic Differential Detection, and certain other programs claim to perform it, too.

But only Untouchable calculates file signatures using not one, but two proprietary algorithms that can't be reverse-engineered.

Only Untouchable guarantees 100% safe recovery of infected files. (Unlike other programs that proudly generate corrupted files, Untouchable knows when the jig is up and doesn't attempt recovery.)

Only Untouchable can provide centralized network virus protection. In fact, Untouchable Network has enough virus alerts and reporting mechanisms to settle the stomach of even the most nervous Netware® administrator.

And only Untouchable is backed by our vaunted 24-hour toll-free technical support and a one-year money-back guarantee.

If you're worried about viruses—and you should be—don't reach for the Maalox.® Reach for the phone. Dial **1-800-677-1848.**

You'll feel better right away.

FIFTH GENERATION SYSTEMS, INC

CIRCLE 286 ON READER SERVICE CARD

Data types in Win16 and Win32

Data type	Win16	Win32
LPVOID	void far * (32-bit pointer)	void * (32-bit pointer)
HANDLE	unsigned int (16-bit value)	unsigned int (32-bit value)
APIENTRY	far pascal	_stdcall

Figure 3: **For the GlobalLock() function, the differences in underlying memory models is hidden within the definitions of these data types.**

tion hiding actually works and protects the existing Windows application source code base so effectively is one of the happiest events of microcomputing history. Certainly, when you consider that the basic Windows memory management API was created before the 80386 was even designed and RISC processors were still a laboratory curiosity, the API's adaptability to such an extreme variety of environments is little short of miraculous. The evolution of the traditional Windows memory API from 1985 to 1992 is summarized in Figure 4.

Of course, the Win32 API brings more to the table than just the traditional Windows memory management functions. In fact, if you are writing a 32-bit application from scratch for Win32 and portability to or from Win16 is not an issue, you may find the prospect of using the traditional functions somewhat distasteful. You can turn instead to the new Win32 memory functions shown in Figure 5, specifically designed for the 32-bit flat, paged memory model.

The mapped-file functions, which are the ones we'll concentrate on in this installment, are particularly interesting because they can dramatically simplify applications. The concept behind mapped files is very simple. The data in a mapped file is associated with a range of addresses in the application's memory space. When the application touches a particular address within the mapped range for the first time, it causes a page fault, and the virtual memory manager will go directly to the file, retrieve the appropriate page-size chunk of data, and then place the data in memory at the correct address.

If the application modifies the contents of a page within the mapped range, the hardware will flag the page as "dirty," and the virtual memory manager will then automatically write that page back to the file when the physical page is needed for something else, the file is unmapped, or the program terminates.

The mapped-file functions are elegant on several levels. The work of file buffering is shifted away from the application onto the operating system's virtual memory manager, which can take full advantage of its "magical" knowledge about the underlying paging hardware and the system's free memory resources. The application can freely access any location in the file at any time with a simple memory access, without worrying about allocating, initializing, refilling, or flushing buffers. A mapped file's in-memory pages are demand-allocated out of the system's global pool, so if the overall load on the system is small, the amount of swapping

that is required for accesses to the mapped file will diminish accordingly. A mapped file can be shared by two or more processes, in effect allowing the processes to communicate through the file "view" and eliminating the redundant allocation of buffers. Best of all, no unnecessary file I/O is ever performed; the only file data that is transferred from disk into memory is the data that the application actually inspects or changes.

Memory Management Under Windows

Windows API Memory Function	Parameters	Windows 1.*x* (real mode)	Windows 3.1 (16-bit protected mode)	Win32/NT (32-bit protected mode)
GlobalAlloc	argument(s)	attributes, number of bytes	attributes, number of bytes	attributes, number of bytes
	results	handle	selector	32-bit offset
GlobalLock	argument(s)	handle	selector	32-bit offset
	results	segment:offset	selector:offset	32-bit offset
GlobalReAlloc	argument(s)	handle, number of bytes, attributes	selector, number of bytes, attributes	32-bit offset, number of bytes, attributes
	results	handle	selector	32-bit offset
GlobalUnlock	argument(s)	handle	selector	32-bit offset
	results	flag	flag	flag
GlobalFree	argument(s)	handle	selector	32-bit offset
	results	handle	selector	32-bit offset

Figure 4: **The traditional Windows memory management API as it has evolved from the real-mode Windows of 1985 to the 32-bit protected-mode Win32 of today.**

USING MAPPED-FILE FUNCTIONS The first step in using the mapped-file functions is to open the file using the familiar Windows functions OpenFile(), CreateFile(), or _lopen(), all of which return a file handle. The file handle is then supplied to the function CreateFileMapping(), along with several other parameters: the maximum number of bytes of the file to map, the access type (read-only or read-write), an optional security attribute, and an optional arbitrary name for the mapping. If CreateFileMapping() succeeds, it returns a handle to a file mapping object. And by the way, as you begin to program for NT and to explore the parts of the Win32 API that are not pres-

Win32 Memory Management Functions

Memory allocation	VirtualAlloc(), VirtualFree(), VirtualProtect(), Virtual Query()
Heap management	HeapCreate(), HeapDestroy(), HeapAlloc(), HeapFree(), HeapSize()
Mapped files	CreateFileMapping(), OpenFileMapping(), MapViewOfFile(), MapViewOfFileEx(), FlushViewOfFile(), UnmapViewOfFIle()

***Figure 5:* These new Win32 functions are specifically designed for the 32-bit flat, paged memory model.**

ent in Win16, you're going to have objects coming out of your ears. Objects are everywhere in NT; they are the fundamental unit of resource management. In fact, the NT microkernel's main mission is to create, destroy, and control access to objects. Objects as seen by an application are just handles representing hidden data structures (controlled by the operating system) that encapsulate ownership and state information for processes, memory, files, semaphores, timers, and so on.

In any event, once your program has a file-mapping object in hand, it calls the function MapViewOfFile() to associate the mapping object with a range of memory addresses. MapViewOfFile() has a number of parameters, including the mapping object handle, an offset and size that allow a subset of the file to be mapped instead of the entire file, and an access type. Note that access rights are specified at three points during the mapping procedure: the opening of the file, the creation of the mapping object, and the mapping of the file view. All of the access rights must be consistent for the file mapping to work.

Assuming that the MapViewOfFile() function succeeds, it returns a pointer and the contents of the file are immediately "visible," using the pointer as an address base. (If the application needs to map the file at specific addresses, it can call MapViewOfFileEx() rather than allowing the system to assign the addresses.) Suppose we want to create a mapped view of the file myfile.dat and zero out the byte at offset 2000h within the file. A skeleton for the required source code would look like this:

```
HANDLE hFile;
HANDLE hMap;
LPSTR lpMap;
.
.
.
hFile = _lopen("myfile.dat". . .);
hMap =
 CreateFileMapping(hFile. . .);
lpMap = MapViewOfFile(hMap. . .);
lpMap[0x2000] = 0;
```

When the program is done accessing the file, it destroys the file mapping by calling UnmapViewOfFile(), releases the file-mapping object by calling CloseHandle(), and finally closes the file itself with _lclose or CloseHandle(). At any point in between, the program can force the in-memory and on-disk contents of the file to be brought into sync with the FlushViewOfFile() function.

Two or more programs can arrange to use a mapped file for interprocess communication in several ways. A mapped-file object handle can be inherited by child processes. A handle can be duplicated and passed to other processes.

HEXVIEW2.C

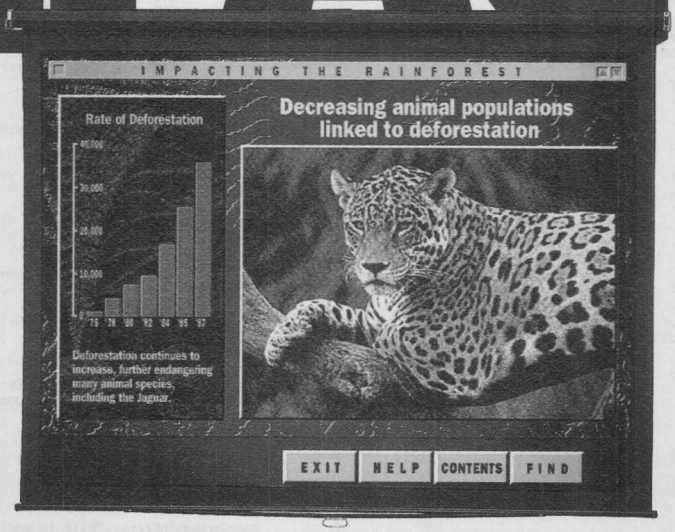
1 of 2

```
// HexView2 - Hex File Viewer for NT/WIN32 Using Mapped File Functions
// Copyright (C) 1992 Ray Duncan
// PC Magazine * Ziff Davis Publications

#define dim(x) (sizeof(x) / sizeof(x[0]))   // returns no. of elements
#define EXENAMESIZE 256                     // max length of path+filename
#define BPL 16                              // bytes per line

#include "stdlib.h"
#include "windows.h"
#include "string.h"
#include "commdlg.h"
#include "hexview.h"

HANDLE hInst;                               // module instance handle
HWND hFrame;                                // handle for frame window
HWND hChild;                                // handle for child window

HFONT hFont;                                // handle for nonprop. font
INT CharX, CharY;                           // character dimensions
INT LPP;                                    // lines per page
INT BPP;                                    // bytes per page
INT ThumbInc;                               // bytes per thumb unit
INT WinWidth;                               // width of frame client area

INT hFile = -1;                             // handle for current file
char szFileName[EXENAMESIZE+1];             // name of current file
LONG FileSize;                              // length of current file
LONG FileIndex;                             // index from thumb tracking

LONG ViewPtr;                               // addr, first line current page
LONG TopAddr;                               // address, top of last page
LONG TopLine;                               // line num, top of last page

HANDLE hMap;                                // handle, file mapping object
LPSTR lpMap;                                // base address, mapping object

.
.  Remaining static variables and tables at start of program
.  are unchanged from previous version of HEXVIEW.  WinMain()
.  and InitApp() routines are unchanged.
.

//
// InitInstance --- instance initialization code for this application.
// Gets information about system nonproportional font.  Creates frame
// window, gets initialization information (if any) from registration
// database.  Positions and sizes window, opens and positions previous
// file if initialization info found.
//
BOOL InitInstance(HANDLE hInstance, INT nCmdShow)
{
    HDC hdc;                                // handle for device context
    TEXTMETRIC tm;                          // font information
    RECT rect;                              // window position & size
    char buff[80];                          // scratch buffer

    hFrame = CreateWindow(                  // create frame window
        szFrameClass,                       // window class name
        szAppName,                          // text for title bar
        WS_OVERLAPPEDWINDOW | WS_VSCROLL,   // window style
        CW_USEDEFAULT, CW_USEDEFAULT,       // default position
        CW_USEDEFAULT, CW_USEDEFAULT,       // default size
        NULL,                               // no parent window
        NULL,                               // use class default menu
        hInstance,                          // window owner
        NULL);                              // unused pointer

    if(!hFrame) return(FALSE);              // error, can't create window

    hdc = GetDC(hFrame);                    // get device context
    hFont = GetStockObject(SYSTEM_FIXED_FONT);
    SelectObject(hdc, hFont);               // realize nonproportional
    GetTextMetrics(hdc, &tm);               // font, get character size,
    CharX = tm.tmAveCharWidth;              // and calculate window width
    CharY = tm.tmHeight + tm.tmExternalLeading;
    WinWidth = (CharX * 75) + GetSystemMetrics(SM_CXVSCROLL);
    ReleaseDC(hFrame, hdc);                 // release device context

    GetWindowRect(hFrame, &rect);           // get window position, size

    // read saved position/size profile for our frame window, if any
    rect.left = GetPrivateProfileInt("Frame", "xul", rect.left, szIni);
    rect.top = GetPrivateProfileInt("Frame", "yul", rect.top, szIni);
```

PC MAGNET

***Figure 6:* Extracts from the source code for the mapped-file version of HEXVIEW. The executable program and complete source code files are available for downloading from PC MagNet.**

THE NEW LITEPRO™ IS THE WORLD'S ONLY ALL-IN-ONE PORTABLE LCD DATA PROJECTOR THAT GIVES YOU INSTANT IMPACT!



Everything you need for electronic presentations.

See that sleek little black box over there? You'd never believe what's packed into it.

A high-quality color LCD panel, an overhead projector, and even a portable computer. All in one.

It's the new In Focus LitePro. And it will give your presentations a whole new level of impact. Instantly.

Just plug it in and turn it on. What could be easier?

Simply connect to most any PC or Mac, and present in real time. Or with the LitePro LS, slide a floppy into the exclusive built-in LiteShow,™ and you're using the world's first electronic slide projector.

Either way, it's the fastest way to present, eliminating the time and costs of outputting to slides and transparencies.

Easy on your back—and your wallet.

The LitePro is very affordable and lightweight considering everything you get—the very latest in LCD technology with thousands of brilliant colors at 640 x 480 resolution.

But we don't stop with the LitePro. In Focus offers you the most complete line of LCD projection products available.

So give us a call today.

Our GSA # is GS00K91AGS5205 PS01.

InFocus® SYSTEMS

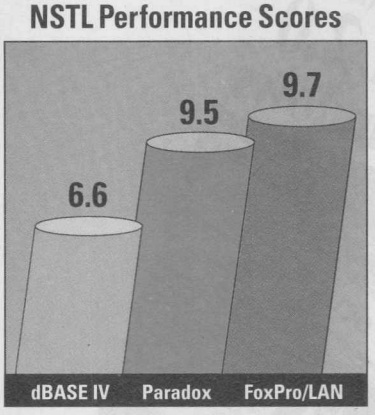
For info and a free book on more effective presentations, call 1-800-327-7231.

HEXVIEW2.C

2 of 2

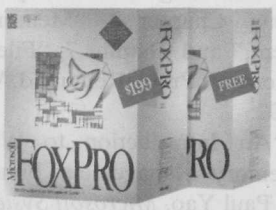
```
    rect.right = GetPrivateProfileInt("Frame", "xlr", rect.right, szIni);
    rect.bottom = GetPrivateProfileInt("Frame", "ylr", rect.bottom, szIni);

    MoveWindow(hFrame, rect.left, rect.top, // force window position, size
        WinWidth, rect.bottom-rect.top, TRUE);

    // get saved filename and file offset for display, if any
    GetPrivateProfileString("File", "filename", "", szFileName,
        EXENAMESIZE, szIni);
    GetPrivateProfileString("File", "fileptr", "", buff, sizeof(buff), szIni);

    if(szFileName[0])                       // if filename and file offset
    {                                       // was saved from previous
        OpenDataFile();                     // execution, open the file
        SetFilePosition(atol(buff));
    }

    ShowWindow(hFrame, nCmdShow);           // make frame window visible
    UpdateWindow(hFrame);                   // force WM_PAINT message
    return(TRUE);                           // return success flag
}

.
. Routines TermInstance(), FrameWndProc(), ChildWndProc(),
. DoCommand(), DoDestroy(), DoClose(), DoVScroll(), DoPaint(),
. DoChildPaint(), DoSize(), DoMenuExit(), DoMenuAbout(),
. AboutDlgProc(), Repaint(), SetFilePosition(), ConfigDisplay(),
. ThumbTrack(), and UpdateFrameProfile() are unchanged from
. previous version of HEXVIEW. Routines ReadDataFile() and
. GetByte() are superfluous and are no longer present.
.

//
// DoMenuOpen -- process File-Open command from menu bar. All
// the hard work is done by the OpenFile common dialog.
//
LONG DoMenuOpen(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    OPENFILENAME ofn;                       // used by common dialogs

    szFileName[0]  = '\0';                  // init filename buffer

    ofn.lStructSize = sizeof(OPENFILENAME); // length of structure
    ofn.hwndOwner = hWnd;                   // handle for owner window
    ofn.lpstrFilter = szFilter[0];          // address of filter list
    ofn.lpstrCustomFilter = NULL;           // custom filter buffer address
    ofn.nFilterIndex = 1;                   // pick default filter
    ofn.lpstrFile = szFileName;             // buffer for path+filename
    ofn.nMaxFile = EXENAMESIZE;             // length of buffer
    ofn.lpstrFileTitle = NULL;              // buffer for filename only
    ofn.lpstrInitialDir = NULL;             // initial directory for dialog
    ofn.lpstrTitle = NULL;                  // title for dialog box
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
    ofn.lpstrDefExt = NULL;                 // default extension

    if(GetOpenFileName(&ofn))               // display open dialog
    {
        OpenDataFile();                     // open file for viewing
        Repaint();                          // force display of data
    }

    return(FALSE);
}

//
// SetWindowCaption -- concatenate the specified filename with the
// application name, then update the frame window's title bar.
// If called with NULL pointer, removes any previous filename from
// the title bar, leaving only the application name.
//
VOID SetWindowCaption(char * szName)
{
    char szTemp[EXENAMESIZE+1];

    if(szName == NULL)                      // if null pointer, just
        SetWindowText(hFrame, szAppName);   // display application name
    else
    {                                       // otherwise...
        strcpy(szTemp, szAppName);          // get application name
        strcat(szTemp, " - ");              // add separator
        strcat(szTemp, szName);             // add filename
        SetWindowText(hFrame, szTemp);      // put result into title bar
    }
}

//
// DisplayLine -- format and display a single line of hex/ASCII dump
// using the device context and relative window line number supplied
// by the caller.  The file data offset is calculated from the
// relative window line and the offset corresponding to the line
// currently at the top of the window.
//
VOID DisplayLine(HDC hdc, INT line)
{
    INT i;
    char buff[256], c, *p;
    LONG x;

    if((ViewPtr + (LONG) line*BPL) < FileSize)
    {
        // format file offset as 8-digit hex number
        p = buff + wsprintf(buff, "%08lX ", ViewPtr + (LONG) line*BPL);

        // format this 16-bytes as hex data
        for(i = 0; i < BPL; i++)
        {
            x = ViewPtr + (LONG)(line * BPL) + i;

            if(x < FileSize)
                p += wsprintf(p, "%02X ", lpMap[x] & 0x00ff);
            else
                p += wsprintf(p, "   ");
        }

        // format same 16-bytes as ASCII, using "." for control characters
        for(i = 0; i < BPL; i++)
        {
            x = ViewPtr + (LONG)(line * BPL) + i;

            if(x < FileSize)
            {
                c = lpMap[x];
                if(c < 0x20)
                    c = '.';
            }
            else c = ' ';
            *p++ = c;
        }

        // append a null byte, then paint the formatted data
        *p = '\0';
        TextOut(hdc, 0, line*CharY, buff, strlen(buff));
    }
}

//
// OpenDataFile -- open specified file for viewing, save handle. The
// filename was previously placed in szFileName by OpenFile common dialog.
//
VOID OpenDataFile(VOID)
{
    char temp[256];             // scratch formatting buffer

    if(hFile != -1)                         // if previous file,
    {                                       // release mapping object
        UnmapViewOfFile(lpMap);             // and view, then close
        CloseHandle(hMap);                  // the file
        _lclose(hFile);
    }

    // remove any previous filename from title bar
    SetWindowCaption(NULL);

    hFile = _lopen(szFileName, OF_READ);    // try and open the new file

    if(hFile == -1)                         // error, no such file
    {
        wsprintf(temp, "Can't open file: %s", szFileName);
        MessageBox(hFrame, temp, szAppName, MB_ICONSTOP | MB_OK);
        return;
    }

    FileSize = _llseek(hFile, 0, 2);        // get size of file

    if(FileSize == 0)                       // error, nothing to display
    {
        MessageBox(hFrame, "File is empty!", szAppName,
            MB_ICONSTOP | MB_OK);
        _lclose(hFile);
        hFile = -1;
        return;
    }

    // create file mapping object with no maximum size
    hMap = CreateFileMapping((HANDLE) hFile,
        (LPSECURITY_ATTRIBUTES) NULL, PAGE_READONLY, 0, 0, (LPSTR) NULL);

    if(hMap == 0)                           // error, no object available
    {
        MessageBox(hFrame, "Can't create file mapping object!",
            szAppName, MB_ICONSTOP | MB_OK);
        _lclose(hFile);
        hFile = -1;
        return;
    }

    // mapping object created, now map view of file onto object
    lpMap = (LPSTR) MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0);

    if(lpMap == 0)                          // error, no view created
    {
        MessageBox(hFrame, "Can't map view of file!",
            szAppName, MB_ICONSTOP | MB_OK);
        CloseHandle(hMap);
        _lclose(hFile);
        hFile = -1;
        return;
    }

    ViewPtr = 0;                            // reset window address pointer
    ConfigDisplay();                        // set up scroll bar
    SetWindowCaption(szFileName);           // put filename in title bar
}
```

Which is really the fastest database?



Microsoft FoxPro 2.0 is the fastest PC DBMS...faster than dBASE IV® 1.5, and faster than Paradox 4.0†

See for yourself.



Upgrade to Microsoft FoxPro 2.0 for $199 and get version 2.5 FREE.

If you use dBASE®, Paradox®, Clipper™ or any other PC database, you should know that FoxPro 2.0 has been consistently rated the fastest database — as recently as an independent September 1992 NSTL lab report†.

And if you use any other database, there's something else you should know about — a way to really see, for yourself, which is the fastest database around. With this special, limited-time offer from Microsoft, you can get Microsoft® FoxPro® database version 2.0 (single user version) for just $199*, with a free** upgrade to version 2.5 for MS-DOS® or for Windows® operating systems as soon as it's released.

Just call before December 31, 1992 and get a great deal on the fastest PC DBMS available. Plus a great deal of support. With Microsoft's full commitment, Microsoft FoxPro is sure to stay at the head of the database pack.

For more information about Microsoft FoxPro's performance, for the name of a dealer near you, or to order direct from Microsoft, call now.

Call (800) 228-7007, Ext. AM9

Microsoft®

Making it easier

* Plus $10.00 freight and applicable sales tax. Reseller prices may vary. **Free, you don't even pay freight. †dBASE IV and Paradox Benchmarks from an independent NSTL Report and *Software Digest* rating report, vol. 9, No. 2. You must currently be a licensed user of a database product to qualify for this upgrade offer. To qualify for this offer at your reseller you must bring proof of purchase with you: 1) the original disk from your program, 2) title page of the manual, or 3) a copy of the sales receipt. If you get your upgrade directly from Microsoft, you will be automatically registered. You must register your copy of Microsoft FoxPro 2.0 to qualify for a free upgrade to version 2.5. You will then be notified when your free upgrade is available. Offer expires December 31, 1992. Offer good only in the 50 United States. Limit one copy of Microsoft FoxPro version 2.0 and one free upgrade to version 2.5 per product license. Please allow 2-4 weeks for delivery upon receipt of this order by Microsoft. In the 50 United States, call (800) 228-7007, Ext. AM9. For information only: In Canada, call (800) 563-9048; outside the 50 United Sates and Canada, call (206) 936-8661. Microsoft, MS-DOS and FoxPro are registered trademarks and Windows is a trademark of Microsoft Corporation. dBASE, dBASE IV, and Paradox are registered trademarks of Borland International, Inc. Clipper is a trademark of Nantucket Corporation. Software Digest is a registered trademark of NSTL Inc. Datapro Company.

If a name was assigned to the mapping object during the call to CreateFileMapping() and the security attributes allow it, other processes can open the mapping object by name with the function OpenFileMapping(). As a special case, there is also a way to use the mapped-file functions for interprocess communication without an associated file, roughly equivalent to the named shared-memory segments of OS/2.

In order to provide a practical example of programming the mapped-file API, I've modified last issue's HEXVIEW utility to produce a new version that eliminates explicit file buffering in favor of file mapping. We don't have space to print the complete listing here, but extracts of the source code relevant to the use of mapped files can be seen in Figure 6, and the complete source code and executable program can be downloaded from PC MagNet, archived as HEXVI2.ZIP. Ironically, the most important parts of the program conversion are the parts you won't find either here or on PC MagNet: The code to allocate and release a file I/O buffer and the routines ReadDataFile() and GetByte() all go away, as they are not needed.

FURTHER READING These references can help you learn more about managing memory in a 32-bit environment:

• *Win32 Programmer's Reference: Overviews.* Chapter 2, "Memory Management," and Chapter 7, "File Mapping." Microsoft Press, Redmond, Washington, 1992.

• "An Introduction to Windows NT Memory Management Fundamentals," by Paul Yao. *Microsoft Systems Journal,* July/August 1992, volume 7, number 4, page 41.

• *Microsoft's 80386/80486 Programming Guide,* 2nd Edition, by Ross Nelson. Microsoft Press, Redmond, Washington, 1991. ISBN: 1-55615-343-0.

• *Programming the 80386,* by John Crawford and Patrick Gelsinger. Sybex Inc., Alameda, California, 1987. ISBN: 0-89588-381-3.

THE IN-BOX Please send your questions, comments, and suggestions to me at any of these electronic mail addresses:

PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan
Internet: duncan@csmc.edu □